Semantic Web Logic Programming Tools

J. J. Alferes, C. V. Damásio, and L. M. Pereira

Centro de Inteligência Artificial - CENTRIA
Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
jja | cd | lmp @di.fct.unl.pt

Abstract. The last two decades of research in Logic Programming, both at the theoretical and practical levels, have addressed several topics highly relevant for the Semantic Web effort, providing very concrete answers to some open questions.
This paper describes succinctly the contributions from the Logic Programming group of Centro de Inteligência Artificial (CENTRIA) of Universidade Nova de Lisboa, as a prelude to a description of our recent efforts to develop integrated standard tools for disseminating this research throughout the interested Web communities. The paper does not intended to be a survey of logic programming techniques applicable to the Semantic Web, and so the interested reader should try to obtain the missing information in the logic programming journals and conferences.

1 Introduction

The eXtensible Markup Language provides a way of organizing data and documents in a structured and universally accepted format. However, the tags used have no predefined meaning. The W3C has proposed the Resource Description Framework (RDF) for exposing the meaning of a document to the Web community of people, machines, and intelligent agents [26].

Conveying the content of documents is just a first step for achieving the full potential of the Semantic Web. Additionally, it is mandatory to be able to reason with and about information spread across the World Wide Web. The applications range from electronic commerce applications, data integration and sharing, information gathering, security access and control, law, diagnosis, B2B, and of course, to modelling of business rules and processes.

Rules provide the natural and wide-accepted mechanism to perform automated reasoning, with mature and available theory and technology. This has been identified as a Design Issue for the Semantic Web, as clearly stated by Tim Berners-Lee et al in [10]:

> *"For the semantic web to function, computers must have access to structured collections of information and sets of inference rules that they can use to conduct automated reasoning."*
>
> *"The challenge of the Semantic Web, therefore, is to provide a language that expresses both data and rules for reasoning about the data and*

that allows rules from any existing knowledge-representation system to be exported onto the Web."

"Adding logic to the Web – the means to use rules to make inferences, choose courses of action and answer questions – is the task before the Semantic Web community at the moment."

Logic Programming is about expressing knowledge in the form of rules and making inferences with these rules. A major advantage of Logic Programming is that it provides an operational reading of rules and a declarative reading with well-understood semantics. In this paper we defend the use of Generalized Extended Logic Programs [29], i.e. logic programs with both (monotonic) explicit negation and (non-monotonic) default negation, as an appropriate expressive formalism for knowledge representation in the Web.

An important feature of Logic Programming is that it is able to deal with negative knowledge, and to express closed world assumptions. Expressing and reasoning with negative knowledge is fundamental for advanced applications but these capabilities are currently lacking in the existing and proposed Web standards. This is clearly identified as a limitation/feature of RDF, in the latest W3C Working Draft of RDF Semantics [24]:

"RDF is an assertional logic, in which each triple expresses a simple proposition. This imposes a fairly strict monotonic discipline on the language, so that it cannot express closed-world assumptions, local default preferences, and several other commonly used non-monotonic constructs."

The introduction of (non-monotonic) default negation brought new theoretical problems to Logic Programming, which were addressed differently by the two major semantics: Well-founded Semantics [21] (WFS) and Stable Model Semantics [22] (SM).

We start this paper by defending the use of Well-founded based semantics as an appropriate semantics for Semantic Web rule engines, and by illustrating its usage. We then proceed, in section 4 to describe our W4 project – Well-Founded Semantics for the WWW – which aims at developing Standard Prolog inter-operable tools for supporting distributed, secure, and integrated reasoning activities in the Semantic Web, and describe the implementations already developed within the project.

The Semantic Web is a "living organism", which combines autonomously evolving data sources/knowledge repositories. This dynamic character of the Semantic Web requires (declarative) languages and mechanisms for specifying its maintenance and evolution. It is our stance that also in this respect Logic Programming is a good choice as a representational language with attending inference and maintenance mechanisms and, in section 5, we briefly describe our recent research efforts for defining and implementing logic programming systems capable of dealing with updates and knowledge-base evolution.

2 The case for Well-founded based Semantics

As mentioned above, in this paper we propound the use of Generalized Extended Logic Programs [29] as an appropriate expressive formalism for knowledge representation in the Web. A Generalized Extended Logic Program is a set of rules of the form:

$$L_0 \leftarrow L_1, \ldots, L_n$$

where literals $L_0, L_1, \ldots, L_n$ are objective literals, say A or $\neg A$, or default negated objective literals, say $not\, A$ or $not\, \neg A$, with A an atom of a given first-order language. Without loss of generality, a non-ground rule stands for all its ground instances. Notice that two forms of negation are available, namely default (or weak) negation not and explicit (or strong) negation $\neg$, and can occur both in the head (L_0) and body ($L_1, \ldots, L_n$) of the rule.

Default negation is non-monotonic and captures what is believed or assumed false (closed-world assumption), whilst explicit negation is monotonic and expresses what is known to be false (open-world assumption). The rationale of the two forms of negation is better grasped with the following example attributed to McCarthy:

Example 1. Suppose a driver intends to cross a railway and must make a decision whether he should proceed or stop. The two major possibilities he has to encode the knowledge in a logic programming language are captured by the rules:

$$1)\ cross \leftarrow \neg train \qquad\qquad 2)\ cross \leftarrow not\, train$$

Rule 1) represents the usual behaviour of a safe driver by stating that he can cross the rail tracks only when he has explicit evidence that a train is not approaching. The second rule represents the situation of a careless driver that advances whenever there is no evidence that a train is approaching (i.e. believes/assumes the train is not approaching).

The introduction of default negation brought new theoretical problems, which were addressed differently by the two major semantics for logic programs: Well-Founded Semantics [21] (WFS) and Stable Model Semantics [22] (SM). We suggest the use of Well-Founded based Semantics as an appropriate semantics for Semantic Web rule engines, by the following reasons:

- The adopted semantics for definite, acyclic and (locally) stratified logic programs, coinciding with Stable Model Semantics.
- Defined for every normal logic program, i.e. with default negation in the bodies, no explicit negation and atomic heads.
- Polynomial data complexity with efficient existing implementations, namely the SLG-WAM engine implemented in XSB [30].
- Good structural properties.
- It has an undefined truth-value.
- Many extensions exist over WFS, capturing paraconsistent, incomplete, and uncertain reasoning.

– Permits update semantics via Dynamic Logic Programs and EVOLP.
– It can be readily "combined" with DBMSs, Prolog, and Stable Models engines.

The minimal Herbrand model semantics for definite logic programs [20] (programs without default and explicit negation) is well-understood and widely accepted. Both Well-Founded Semantics and Stable Model Semantics coincide with the minimal Herbrand model semantics for definite logic programs.

A major advantage of WFS is that it is possible to assign a unique model to every normal logic program, in constrast to SM semantics. The same applies to the several extensions of WFS treating explicit negation, supporting paraconsistent reasoning forms [16], which we shall discuss in the following section.

The existence of an undefined logical value is fundamental for Semantic Web aware inference engines. On the one hand, in a distributed Web environment with communication failures and non-ignorable response times, a "remote" logic inference can be assumed undefined, while the computation proceeds locally. If the remote computation terminates and returns an answer, then the undefined truth-value can be logically updated to true or false. On the other hand, rule bases in the Web will naturally introduce cycles through default negation. Well-Founded Semantics deals with these cycles through default negation by assigning the truth-value undefined to the literals involved. In this particular situation, Stable Models may not exist or may explode.

The computation of the Well-Founded Model is tractable, contrary to Stable Models, and efficient implementations exist, notably the XSB Prolog engine [30]. XSB resorts to tabling techniques, ensuring better termination properties and polynomial data complexity. Tabling is also a good way to address distributed query evaluation of definite and normal logic programs. The XSB Prolog supports a full first-order syntax, which is not fully available in the state-of-the art Stable Model engines [9, 31]. Moreover, the latests XSB Prolog 2.6 distribution is integrated with the SModels system, and thus applications can better exploit both Well-Founded and Stable Model semantics.

In summary, Well-Founded Semantics can be seen as the light-inference basic mechanism for deploying **today** complex Semantic Web rule-based applications, and Stable Model Semantics a complementary semantics for addressing other complex reasoning forms.

3 Knowledge Representation with Explicit Negation

In this section we illustrate the use of explicit and default negation for representing ontological knowledge, which may be contradictory and/or incomplete. The ability to deal and pinpoint contradictory information is a desirable feature of Semantic Web rule system, since it is very natural to obtain conflicting information from different sources. Classical logic assigns no model to an inconsistent theory, and therefore it is not fully appropriate as a general knowledge representation formalism for the Semantic Web. This limitation is inherited by

the classical logic based formalisms like RDF(S) [26, 24, 12], DAML+OIL [13], and OWL [17].

An interesting example is the case of taxonomies. The example is a natural one since our common sense knowledge of the animal world is rather limited, and new species are discovered frequently. We present some examples showing the capabilities of our own Generalized Paraconsistent Well-founded Semantics with Explicit Negation, $WFSX_P$ for short. For a full account and all the formal details, the reader is referred to [3, 14, 16].

Example 2. Consider the following common-sense rules for identifying birds and mammals:

- Oviparous warm-blooded animals with a bill are birds;
- Hairy warm-blooded animals are mammals;
- Birds are not mammals and vice-versa;
- Birds fly;
- Mammals nurse their offspring.

This chunk of knowledge can be represented by the following extended logic program rules:

$$
\begin{array}{l}
bird(X) \leftarrow bill(X), warm\_blood(X), oviparous(X). \\
\neg bird(X) \leftarrow mammal(X). \\
flies(X) \leftarrow bird(X).
\end{array}
$$

$$
\begin{array}{l}
mammal(X) \leftarrow hair(X), warm\_blood(X). \\
\neg mammal(X) \leftarrow bird(X). \\
nurses(X) \leftarrow mammal(X).
\end{array}
$$

If the information regarding dogs and ducks is correctly filled in one gets the expected results. We just add to the program the set of facts:

$$hair(dog).\ warm\_blood(dog).\quad bill(duck).\ warm\_blood(duck).\ oviparous(duck).$$

The model of the above program under $WFSX_P$ entails the following expected conclusions:

$$
\left\{
\begin{array}{l}
mammal(dog), nurses(dog), \neg bird(dog), not\, bird(dog), not\, flies(dog), \\
bird(duck), flies(duck), \neg mammal(duck), not\, mammal(duck), \\
not\, nurses(duck)
\end{array}
\right\}
$$

Now on a trip to Australia the user discovers there are some creatures named platypus which lay eggs, have warm blood, sport a bill, and are hairy! A nice contradiction is obtained from the program containing the facts:

$$hair(platypus).\ \ warm\_blood(platypus).\ \ bill(platypus).\ \ oviparous(platypus).$$

The model entails the following new conclusions:

$$\left\{\begin{array}{l} mammal(platypus), \neg mammal(platypus), \\ not\, mammal(platypus), not\, \neg mammal(platypus), \\ nurses(platypus), not\, nurses(platypus), not\, \neg nurses(platypus), \\ bird(platypus), \neg bird(platypus), not\, bird(platypus), not\, \neg bird(platypus), \\ flies(platypus), not\, flies(platypus), not\, \neg flies(platypus) \end{array}\right\}$$

The remarkable points about this example are manifold. First, contradictory information can coexist with safe one without interfering with each other; in particular, we must not relinquish the information about dogs and ducks. Second, we can detect a contradiction both about the *mammal* and the *bird* predicates (both *mammal* and $\neg mammal$ hold, as well as *bird* and $\neg bird$), its consequences are propagated, and we are aware that the knowledge about platypuses regarding nursing and flying capabilities depends on contradiction. This is recognized by noticing that $nurses(platypus)$ and $not\, nurses(platypus)$ hold, while $\neg nurses(platypus)$ is absent from the model [15]. Third, the right solution is covered by the program's model: platypus are mammals, do not fly, and nurse their progeny. Finally, it is unsound to have a heuristic rule saying to drop all objective (or default) knowledge. For platypus we want to retain that they nurse their descendants but discard the $fly(platypus)$ conclusion.

The rationale of $WFSX_P$ is to non-trivially extract the maximum number of conclusions from contradictory information. This provides the user with the information necessary to decide what to do, since all possible scenarios are taken into account. The user is warned about some potential problems, and is up to him to take the right decision. This is possible due to the adoption of the Coherence Principle, which relates both forms of negation: *"If something is known to be false then it should be believed false: if $\neg A$ holds then not A should hold; if A holds then not $\neg A$ should hold".*

If A and $\neg A$ hold, then by coherence, one should have $not\, \neg A$ and $not\, A$. This produces a localized explosion of consequences which are propagated by the semantics only to the dependant literals, and not to the whole model. The same semantics can be exploited to represent taxonomies with exceptions, expressing general absolute (i.e. non-defeasible) rules, defeasible rules, exceptions to defeasible rules and to other exceptions, explicitly making preferences among defeasible rules. We assume that in the presence of contradictory defeasible rules we prefer the one with most specific information.

Example 3. Consider the following statements, corresponding to the hierarchy depicted in Figure 1:

(1) Mammals are animals.
(2) Bats are animals.
(3) Birds are animals.
(4) Penguins are birds.
(5) Dead animals are animals.
(6) Normally animals don't fly.
(7) Normally bats fly
(8) Normally birds fly
(9) Normally penguins don't fly
(10) Normally dead animals don't fly

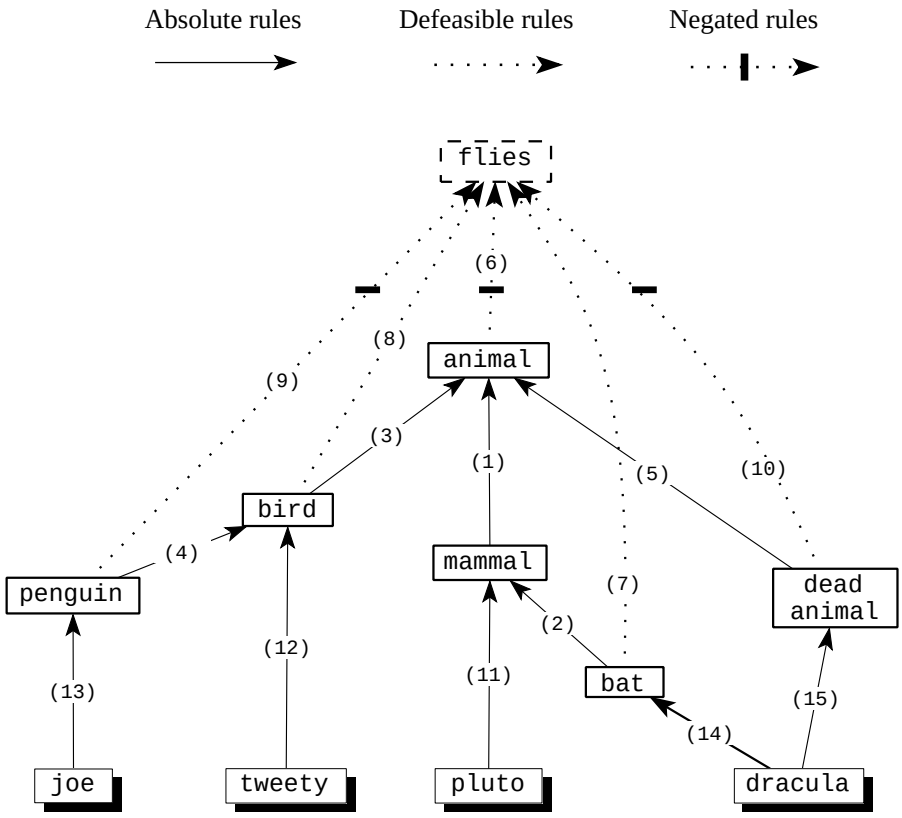

Fig. 1. A non-monotonic hierarchical taxonomy

the following individuals

(11) Pluto is a mammal.
(12) Tweety is a bird.
(13) Joe is a penguin.
(14) Dracula is a bat.
(15) Dracula is a dead animal.

and the preferences

(16) Dead bats do not fly though bats do.
(17) Dead birds do not fly though birds do.
(18) Dracula is an exception to the above preferences.

The above hierarchy can be represented by the program:

(1) $animal(X) \leftarrow mammal(X)$
(2) $mammal(X) \leftarrow bat(X)$
(3) $animal(X) \leftarrow bird(X)$
(4) $bird(X) \leftarrow penguin(X)$
(5) $animal(X) \leftarrow dead\_animal(X)$

(6) $\neg flies(X) \leftarrow animal(X), \neg flying\_animal(X), not\ flies(X)$
$\neg flying\_animal(X) \leftarrow not\ flying\_animal(X)$
(7) $flies(X) \leftarrow bat(X), flying\_bat(X), not\ \neg flies(X)$
$flying\_bat(X) \leftarrow not\ \neg flying\_bat(X)$
(8) $flies(X) \leftarrow bird(X), flying\_bird(X), not\ \neg flies(X)$
$flying\_bird(X) \leftarrow not\ \neg flying\_bird(X)$
(9) $\neg flies(X) \leftarrow penguin(X), \neg flying\_penguin(X), not\ flies(X)$
$\neg flying_penguin(X) \leftarrow not\ flying\_penguin(X)$
(10) $\neg flies(X) \leftarrow dead\_animal(X), \neg flying\_dead(X), not flies(X)$
$\neg flying\_dead(X) \leftarrow not\ flying\_dead(X)$

(11) $mammal(pluto) \leftarrow$
(12) $bird(tweety) \leftarrow$
(13) $penguin(joe)$
(14) $bat(dracula) \leftarrow$
(15) $dead\_animal(dracula) \leftarrow$

with the implicit hierarchical preference rules (prefer most specific information):

$$flying\_animal(X) \leftarrow bat(X), flying\_bat(X)$$
$$flying\_animal(X) \leftarrow bird(X), flying\_bird(X)$$
$$\neg flying\_bird(X) \leftarrow penguin(X), \neg flying\_penguin(X)$$
$$flying\_dead(X) \leftarrow bat(X), flying\_dead\_bat(X).$$

and the explicit problem statement preferences:

(16) $\neg flying\_bat(X) \leftarrow dead\_animal(X), bat(X), \neg flying\_dead\_bat(X)$
$\neg flying\_dead\_bat(X) \leftarrow not\ flying\_dead\_bat(X)$
(17) $\neg flying\_bird(X) \leftarrow dead\_animal(X), bird(X), \neg flying\_dead\_bird(X)$
$\neg flying\_dead\_bird(X) \leftarrow not\ flying\_dead\_bird(X)$
(18) $flying\_dead\_bat(dracula) \leftarrow$

The model of this programs is non-contradictory, and we get the expected results, namely that Pluto and Joe don't fly, and that Dracula and Tweety do fly.

The above rules can be automatically generated from the description of the hierarchies, as shown elsewhere [6]. As for WFS, the advantage of $WFSX_P$ is that the computation of the well-founded model is tractable, is defined for every Generalized Extended Logic Program in particular for contradictory programs. Furthermore, we are able to detect dependencies on contradiction just by looking at the model. Generalized Answer Set Semantics based [23, 29] is an extension of Stable Model Semantics for Generalized Extended Logic Programs, and is very appropriate for the declarative representation of complex problems, but inherits the same problems of Stable Model Semantics; moreover it explodes when faced with contradiction.

4 The W4 project: Well-Founded Semantics for the WWW

The W4 project aims at developing Standard Prolog inter-operable tools for supporting distributed, secure, and integrated reasoning activities in the Semantic Web. The results of the W4 project are expected to contribute to the recently approved REWERSE European Network of Excellence. The long-term objectives are:

- Development of Prolog technology for XML, RDF, and RuleML.
- Development of a General Semantic framework for RuleML, including default and explicit negation, supporting uncertain, incomplete, and paraconsistent reasoning.
- Development of distributed query evaluation procedures for RuleML, based on tabulation, according to the previous semantics.
- Development of Dynamic Semantics for evolution/update of Rule ML knowledge bases.
- Integration of different semantics in RuleML (namely, Well-Founded Semantics, Answer Sets, Fuzzy Logic Programming, Annotated Logic Programming, and Probabilistic Logic Programming).

We have started the implementation efforts from the previously described theoretical work and implementations, and the RuleML [25] language proposal. A full RuleML compiler is already available for an extension of the hornlog frament of RuleML (see [32]). The W4 RuleML compiler supports default and explicit negation both in the heads and in the bodies of rules, as well as assert statements of EVOLP programs (see section 5). The semantics implemented is Paraconsistent Well-founded Semantics with Explicit Negation. We now shortly illustrate the use of the W4 RuleML with an example session:

Example 4. Consider the taxonomy of Example 3 encoded in RuleML format. E.g., one of the rules used for capturing the sentence "Normally, bats fly" is:

$$flies(X) \leftarrow bat(X), flying\_bat(X), not\ \neg flies(X).$$

with the following corresponding RuleML encoding:

```
<imp>
 <_head>
   <atom> <_opr><rel>flies</rel></_opr> <var>X</var> </atom>
 </_head>
 <_body>
   <and>
     <atom> <_opr><rel>bat</rel></_opr> <var>X</var> </atom>
     <atom> <_opr><rel>flying bat</rel></_opr> <var>X</var> </atom>
     <not><neg>
         <atom> <_opr><rel>flies</rel></_opr> <var>X</var> </atom>
     </neg></not>
```

```
    </and>
  </_body>
</imp>
```

The whole rule base is loaded as follows:

```
| ?- loadRules( ruleML( 'taxonomy.ruleml' ) ).
yes
```

The same predicate is capable of reading ordinary Prolog and NTriple files. After loading its rule bases, the user can start querying them, a tuple-at-a-time, with the `demo/2` predicate. The first argument is the name of a loaded rule base and the second the query in the usual Prolog syntax, extended with the unary operators `not` and `neg` for representing default and explicit negation, respectively:

```
| ?- demo( animals, flies(X) ).
X = Dracula; X = Tweety;
no
| ?- demo( animals, neg flies(X) ).
X = Joe; X = Pluto;
no
| ?- demo( animals, ( animal(X), not flies(X) ) ).
X = Pluto; X = Joe;
no
```

The `demo/2` predicate invokes a meta-interpreter that implements $WFSX_P$ semantics via a program transformation into normal logic programming under WFS, making use of the tabling primitives of XSB-Prolog. The predicate `queryRules/3` allows the user to collect all the answers to a query in a list, or write them in XML format to an output stream. The first argument is the rule based being queried, the second is a list of terms of the form `query( Goal, Label, ListofVars)` with the several queries to issue, and finally the last argument is either a variable or an output stream.

```
| ?- queryRules(animals, [query( flies(X), q1, [animal=X] )], Ans).
Ans = [[answer(q1,[animal = Dracula]),answer(q1,[animal = Tweety])]]

| ?- queryRules(animals, [query( flies(X), q1, [animal=X] ),
                          query( neg flies(X), q2, [non=X])], Ans).
Ans = [[answer(q1,[animal = Dracula]),answer(q1,[animal = Tweety])],
       [answer(q2,[non = Joe]),answer(q2,[non = Pluto])]]

| ?- queryRules(animals, [query( flies(X), q1, [animal=X] )],userout).
<answers>
 <_answer><_rlab><ind>q1</ind></_rlab>
          <_subst><var>animal</var><ind>Dracula</ind></_subst>
 </_answer>
 <_answer><_rlab><ind>q1</ind></_rlab>
          <_subst><var>animal</var><ind>Tweety</ind></_subst>
 </_answer>
</answers>
```

```
| ?- queryRules(animals, [query( flies(X), q1, [animal=X]),
                          query( neg flies(X) , q2, [non=X])], userout).
<answers>
 <_answer><_rlab><ind>q1</ind></_rlab>
          <_subst><var>animal</var><ind>Dracula</ind></_subst>
 </_answer>
 <_answer><_rlab><ind>q1</ind></_rlab>
          <_subst><var>animal</var><ind>Tweety</ind></_subst>
 </_answer>
 <_answer><_rlab><ind>q2</ind></_rlab><_subst>
          <var>non</var><ind>Joe</ind></_subst>
 </_answer>
 <_answer><_rlab><ind>q2</ind></_rlab><_subst>
          <var>non</var><ind>Pluto</ind></_subst>
 </_answer>
</answers>
```

Mark that the answer may be labelled with user-provided labels in order to identify the corresponding query, and variables can be given user-understandable names. The format of answers is not specified in the RuleML proposal.

The W4 RuleML compiler supports several rulebases, imported from RuleML files, Prolog files, or NTriples files. A converter from Prolog syntax to RuleML syntax and from RuleML syntax to Prolog syntax is included. An experimental RDF(S) engine is also provided, and makes extensive use of the tabling facilities of the XSB Prolog engine. By exploiting the NMR features of the new XSB Prolog 2.6, support will be provided for Stable Models and Answer Set Semantics. The package was originally developed for XSB Prolog 2.5, but porting to other Prolog systems is foreseen.

There are some open issues, namely the definition of remote Goal invocation method via the exchange of SOAP messages, and the selection of distributed query evaluation algorithms and corresponding protocols. A standard integration of RuleML with ontologies is still lacking. Further applications, testing, and evaluation is required for the construction of practical systems.

5 Updates and the Evolution of Rule Bases

One of the features for which we developed research work, and corresponding implementations is that of updates and evolution of rule-based knowledge bases. While logic programming can be seen as a good representation language for static knowledge, as we have just shown, if we are to move to a more open and dynamic environment typical of, for example, the agency paradigm, we must consider ways and means of representing and integrating knowledge updates from external sources, but also inner source knowledge updates (or self updates). In fact, an agent not only comprises knowledge about each state, but also knowledge about the transitions between states. The latter may represent the agent's

knowledge about the environment's evolution, coupled to its own behaviour and evolution rules. Similar arguments apply to the Semantic Web. In it, knowledge is stored in various autonomous sources or repositories, which evolve with time, thus exhibiting a dynamic character. Declarative languages and mechanisms for specifying the Semantic Web's evolution and maintenance are in order, and we have recently worked towards this goal.

To address these concerns we first introduced *Dynamic Logic Programming (DLP)* [5] ([19] addressed similar concerns). According to DLP, knowledge is given by a linearly ordered sequence of generalized extended logic programs that represent distinct and dynamically changing states of the world. Each of the states may contain mutually contradictory and overlapping information. The semantics of DLP ensures that all previous rules remain valid (by inertia) so long as they are not contradicted by newer (prevailing) rules.

We have developed two implementations of DLP[1]:

- One of them implements exactly the semantics defined in [5] which is a stable models based semantics. This implementation is provided as a pre-processor of sequences of generalized programs into programs that run under the DLV system [9] for computing the stable models.
- The other implementation is based on a generalization of the well-founded semantics [21] for sequences of programs, which is sound though not complete with respect to the semantics in [5]. The advantages of using a well-founded based semantics rather than a stable models based one can be found in section 2. This implementation consists of a meta-interpreter of sequences of programs, and runs under XSB-Prolog [30]. With it, one can consult sequences of generalized programs, as well as update the running sequence with with set or another of generalized rules. Queries to literals can be posed to the current (latest) state, or to any other previous state. It is also possibly to ask in which of the states some literal holds.

Recently we have integrated a mechanism of preferences [7], which generalizes the preferences of [11] to sequences of programs. The implementation of updates with preferences is based on a pre-processor into DLV programs, according to a transformation defined in [4].

To cope with updates of knowledge coming from various sources, we extended DLP and developed *Multi-dimensional Dynamic Logic Programming - (MDLP)* [28]. DLP allows to encode a single update dimension, where this dimension can either be time, hierarchy strength of rules, priorities, etc. With MDLP more than one of these dimensions can be dealt within a single framework (allowing e.g. to model the evolution over time of hierarchically organized sets of rules). The MDLP implementation is enacted as a meta-interpreter running under XSB-Prolog. With it, Directed Acyclic Graphs (DAGs) of programs can be consulted (with a special syntax for representing the graph), and queries can be put to any of the programs in the DAG.

[1] All implementations mentioned in this section can be found at:
`http://centria.di.fct.unl.pt/~jja/`

With these languages and implementations logic programs can describe well knowledge states and also sequences and DAGs of updating knowledge states. It's only fit that logic programs be utilized to describe the transitions between knowledge states as well. This can be achieved by associating with each state a set of transition rules to obtain the next state. However, till recently, *LP* had sometimes been considered less than adequate for modelling the dynamics of knowledge change over time, because typical update commands are defined by it. To overcome this limitation, we have introduced and implemented the language LUPS [8] (related languages are EPI [18] and KABUL [27]).

LUPS is a logic programming command language for specifying logic program updates. It can be viewed as a language that declaratively specifies how to construct a Dynamic Logic Program by means of successive update commands. A sentence U in LUPS is a set of simultaneous update commands that, given a pre-existing sequence of logic programs, whose semantics corresponds to our knowledge at a given state, produces a new DLP with one more program, corresponding to the knowledge that results from the previous sequence after performing all the simultaneous update commands. A program in LUPS is a sequence of such sentences. The most simple LUPS command is **assert** *Rule*, that simply asserts a rule in the next program of the running sequence. Other more elaborate commands of LUPS take care of retraction of rules, persistent assertion of rules, cancellation of persistent assertions, event assertion and persistent event assertion. For example, the command for persistent rule assertion is of the form **always** *Rule* **when** *Conds*, which from the moment it is given, till cancelled, whenever *Conds* are true *Rule* is asserted. As with DLP, two implementations have also been developed for LUPS, one stable models based, running as a pre-processor into DLV programs, and another running as a meta-interpreter in XSB-Prolog.

More recently, we have worked on a general language, christened EVOLP (after EVOlving Logic Programs) [1], that integrates in a simple way the concepts of both DLP and LUPS, through a language much closer to that of traditional logic programs than the one of LUPS. EVOLP generalizes logic programming to allow specification of a program's own evolution as well as evolution due to external events, in a single unified way, by permitting rules to indicate assertive conclusions in the form of program rules. EVOLP rules are simply generalized LP rules plus the special predicate assert/1, which can appear both in heads or bodies of rules. The argument of assert/1 can be a full-blown EVOLP rule, thus allowing for the nesting of rule assertions within assertions to make it possible for rule updates to be themselves updated down the evolution line. The meaning of a sequence of EVOLP rules is given by sequences of models. Each sequence determines a possible evolution of the knowledge base. Each model determines what is true after a number of evolution steps (i.e. a state) in the sequence: a first model in a sequence is built by "computing" the semantics of the first EVOLP program, where assert/1 is as any other predicate; if $assert(Rule)$ is true at some state, then the program must be updated with *Rule* in the next

state; this updating, and the "computation" of the next model in the sequence, is performed as in DLP.

The current implementation of EVOLP is a meta-interpreter that runs under XSB-Prolog. With it, it is possible to consult sequences of sets of EVOLP rules, as well as update the running sequence with a new set of rules. Queries to literals can be made in the current (later) state, or in any other previous state or interval of states. It is also possibly to ask in which of the states is some literal true. We are now in the process of integrating the EVOLP implementation (which, as mentioned above, encompasses both the features from DLP and LUPS) into the W4 RuleML compiler described in section 4, that already supports EVOLP's syntax . This work, which we expect to finish soon, will allow the usage of EVOLP for taking care of the evolution and maintenance of RuleML rule bases in the Semantics Web.

We have applied these languages, and used the above mentioned implementations, in various domains, such as: actions, agents' architecture, specification of agents' behaviours, software specification, planning, legal reasoning, and active databases. References to this work, as well as the running examples, can be found in the URL above.

To illustrate the expressiveness of these languages, we briefly illustrate here how EVOLP can be employed to model an evolving personal assistant agent for email management able to: Perform basic actions of sending, receiving, deleting messages; Storing and moving messages between folders; Filtering spam messages; Sending automatic replies and forwarding; Notifying the user of special situations. All of this dependent on user specified criteria, and where the specification may change dynamically. More details on this application can be found in [2]. In this application messages are stored via the basic predicates $msg(Identifier, From, Subject, Body, TimeStamp)$ and $in(Identifier, Folder)$ for specifying in which folder the message is stored. New messages are simply events of the form $newmsg(msg(Identifier, From, Subject, Body)$. Basic actions can be easily modelled with EVOLP. For example, for dealing with incoming messages, all we have to specify is that any new message, arriving at time T, should be stored in the inbox folder, unless it is marked for deletion. If a message is marked to be deleted then it should not be stored in any folder. This can be modelled by the EVOLP rules:

$$\begin{aligned} assert(msg(M,F,S,B,T)) &\leftarrow newmsg(M,F,S,B), time(T), not\, delete(M) \\ assert(in(M,inbox)) &\leftarrow newmsg(M,F,S,B), not\, delete(M) \\ assert(not\, in(M,inbox)) &\leftarrow delete(M), in(M,F) \end{aligned}$$

Rules for filtering spam can then be added, as updates to the program, in a simple way. For example, if one wants to filter, and delete, messages containing the word "credit" in the subject, we simply have to update our program with:

$$delete(M) \leftarrow newmsg(M,F,S,B), spam(F,S,B)$$

$$spam(F,S,B) \leftarrow contains(S, credit)$$

Note that this definition of spam can later be updated, EVOLP ensuring that conflicts between older and newer rules are automatically resolved. For example, if later one wants to update the definition of spam, by stating that messages coming from one's accountant should not be considered as spam, all one has to do is to update the program with the rule:

$$not\ spam(F, S, B) \leftarrow contains(F, my\_accountant)$$

With this update, EVOLP ensures that messages from the accountant are not considered spam, even if they contain the word "credit" in the subject, and the user doesn't have to worry about guaranteeing, manually, the consistency of later rules with previous ones.

As an example of a more complex rule, consider that the user is now organizing a conference, and assigns papers to referees. Suppose further that he wants to automatically guarantee that, after receipt of a referee's acceptance, any message about an assigned paper is forwarded to the corresponding referee. In EVOLP terms, this means that if a message is received from the referee accepting to review a given paper, then a rule should be asserted stating that new messages about that paper are to be sent to that referee:

$$\begin{aligned} assert(send(R, S, B) \leftarrow newmsg(M, F, S, B), contains(S, PId), \\ assign(PId, R) \quad) \\ \leftarrow newmsg(M, R, PId, B), contains(B, \text{`accept'}) \end{aligned}$$

For an illustration of more elaborate rules, showing other features of EVOLP, such as the possibility of dynamically changing the policies of the agent triggered by internal or external conditions, for commands that span over various states, etc, the reader is referred to [2].

6 Conclusion

In our opinion, Well-Founded Semantics should be a major player in RuleML, properly integrated with Stable Models. A full-blown theory is available for important extensions of standard WFS/SMs, addressing many of the open issues of the Semantic Web. Most extensions resort to polynomial program transformations, namely those for evolution and update of knowledge bases. They can handle uncertainty, incompleteness, and paraconsistency. Efficient implementation technology exists, and important progress has been made in distributed query evaluation. An open, fully distributed, architecture is being elaborated and proposed.

Acknowledgments

The work summarized here has been developed by us in collaboration with João Alcântara, António Brogi, Pierangelo Dell'Acqua João Leite, Teodor Przymusinski, Halina Przymusinska, and Paulo Quaresma (cf. publications below). We would like to thank all of them for this joint work.

Part of the work on logic programming updates has been supported by projects FLUX "Flexible Logical Updates" (POSI/SRI/40958/2001) , funded by FEDER, and TARDE "Tabulation and Reasoning in a Distributed Prolog Environment" (EEI/12097/1998), funded by FCT/MCES.

References

1. J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Evolving logic programs. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02)*, volume 2424 of *LNAI*, pages 50–61. Springer-Verlag, 2002.
2. J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Logic programming for evolving agents. In M. Klusch, A. Omicini, and S. Ossowski, editors, *Proceedings of the 7th International Cooperative Information Agents (CIA'03)*, LNAI. Springer-Verlag, 2003.
3. J. J. Alferes, C. V. Damásio, and L. M. Pereira. A logic programming system for non-monotonic reasoning. *Special Issue of the Journal of Automated Reasoning*, 14(1):93–147, 1995.
4. J. J. Alferes, P. Dell'Acqua, and L. M. Pereira. A compilation of updates plus preferences. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02)*, volume 2424 of *LNAI*, pages 62–73. Springer-Verlag, 2002.
5. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *The Journal of Logic Programming*, 45(1–3):43–70, September/October 2000. A short version appeared in A. Cohn and L. Schubert (eds.), *KR'98*, Morgan Kaufmann.
6. J. J. Alferes and L. M. Pereira. *Reasoning with logic programming*, volume 1111 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1996.
7. J. J. Alferes and L. M. Pereira. Updates plus preferences. In M. O. Aciego, I. P. de Guzman, G. Brewka, and L. M. Pereira, editors, *Proceedings of the 7th European Conference on Logics in Artificial Intelligence (JELIA'00)*, volume 1919 of *LNAI*, pages 345–360. Springer-Verlag, 2000.
8. J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. Przymusinski. LUPS: A language for updating logic programs. *Artificial Intelligence*, 132(1 & 2), 2002. A short version appeared in M. Gelfond, N. Leone and G. Pfeifer (eds.), *LPNMR'99*, Springer LNAI 1730.
9. The DLV Project: A Disjunctive Datalog System (and more). `http://www.dbai.tuwien.ac.at/proj/dlv/`, 2000.
10. Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, May 2001.
11. G. Brewka and T. Eiter. Preferred answer sets for extended logic programs. *Artificial Intelligence*, 109, 1999. A short version appeared in A. Cohn and L. Schubert (eds.), *KR'98*, Morgan Kaufmann.
12. D. Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema, 23 January 2003 (work in progress). `http://www.w3.org/TR/2003/WD-rdf-schema-20030123/`.
13. D. Connolly D., F. van Harmelen, I. Horrocks, D.L. McGuinness, P.F. Patel-Schneider, and L.A. Stein. DAML+OIL (March 2001) Reference Description. `http://www.w3.org/TR/daml+oil-reference`.

14. Carlos Viegas Damásio and Luís Moniz Pereira. Default negated conclusions: why not ? In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Proc. of the 5th International Workshop on Extensions of Logic Programming (ELP'96)*, number 1050 in LNAI, pages 103–117, 1996.
15. Carlos Viegas Damásio and Luís Moniz Pereira. A paraconsistent semantics with contradiction support detection. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, volume 1265 of *Lecture Notes in Artificial Intelligence*, pages 224–243, Castelo de Dagstuhl, Alemanha, July 1997. Springer.
16. Carlos Viegas Damásio and Luís Moniz Pereira. A survey of paraconsistent semantics for logic programas. In D. Gabbay and P. Smets, editors, *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, volume 2, Reasoning with Actual and Potential Contradictions. Coordenado por P. Besnard e A. Hunter, pages 241–320. Kluwer Academic Publishers, 1998.
17. M. Dean, G. Schreiber, F. van Harmelen, J. Hendler, I. Horrocks, D.L. McGuinness, P.F. Patel-Schneider, and L.A. Stein. OWL Web Ontology Language Reference, 18 August 2003 (Candidate Recommendation). `http://www.w3.org/TR/2003/CR-owl-ref-20030818/`.
18. T. Eiter, M. Fink, G. Sabbatini, and H Tompits. A framework for declarative update specifications in logic programs. In *IJCAI'01*. Morgan-Kaufmann, 2001.
19. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On properties of update sequences based on causal rejection. *Theory and Practice of Logic Programming*, 2(6), 2002.
20. M. Van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of ACM*, 4(23):733–742, 1976.
21. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
22. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *5th International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.
23. M. Gelfond and V. Lifschitz. Logic programs with classical negation. In Warren and Szeredi, editors, *7th International Conference on Logic Programming*, pages 579–597. MIT Press, 1990.
24. P. (Editor) Hayes. RDF semantics, 23 January 2003 (work in progress). `http://www.w3.org/TR/2003/WD-rdf-mt-20030123/`.
25. The Rule Markup Initiative. `http://www.ruleml.org/`.
26. O. Lassila and R. Swick. Resource description framework (RDF) model and syntax specification, 22 February 1999. `http://www.w3.org/TR/REC-rdf-syntax/`.
27. J. A. Leite. *Evolving Knowledge Bases*. IOS Press, 2003.
28. J. A. Leite, J. J. Alferes, and L. M. Pereira. Multi-dimensional dynamic knowledge representation. In T. Eiter, W. Faber, and M. Truszczynski, editors, *Proceedings of the 6th International Conference on Logics Programming and Non-Monotonic Reasoning (LPNMR'01)*, volume 2173 of *LNAI*, pages 365–378. Springer-Verlag, 2001.
29. V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasoning (preliminary report). In B. Nebel, C. Rich, and W. Swartout, editors, *Proceedings of the 3th International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*. Morgan-Kaufmann, 1992.
30. The XSB Logic Programming Systems. `http://xsb.sourceforge.net`, 2003.
31. The smodels system. `http://www.tcs.hut.fi/Software/smodels/`, 2000.
32. The W4 RuleML compiler. `http://centria.fct.unl.pt/~cd/projectos/w4/`.